Preprint
UCRL-JC-147862

# Memory Insensitive Simplification for View-Dependent Refinement

*P. Lindstrom*

This article was submitted to
IEEE Visualization 2002, Boston, MA, October 27 – November 1, 2002

**April 3, 2002**

*U.S. Department of Energy*

Lawrence
Livermore
National
Laboratory

# Memory Insensitive Simplification for View-Dependent Refinement

Peter Lindstrom

*Center for Applied Scientific Computing*
*Lawrence Livermore National Laboratory*

## Abstract

We present an algorithm for end-to-end out-of-core simplification and view-dependent visualization of large surfaces. The method consists of three phases: (1) memory insensitive simplification; (2) memory insensitive construction of a level-of-detail hierarchy; and (3) run-time, output sensitive, view-dependent rendering and navigation of the mesh. The first two off-line phases are performed entirely on disk, and use only a small, constant amount of memory, whereas the run-time component relies on memory mapping to page in only the rendered parts of the mesh in a cache coherent manner. As a result, we are able to process and visualize arbitrarily large meshes given a sufficient amount of disk space—a constant multiple of the size of the input mesh.

Similar to recent work on out-of-core simplification, our memory insensitive method uses vertex clustering on a uniform octree grid to coarsen a mesh and create a hierarchy, and a quadric error metric to choose vertex positions at all levels of resolution. We show how the quadric information can be used to concisely represent vertex position, surface normal, error, and curvature information for anisotropic view-dependent coarsening and silhouette preservation.

The focus of this paper is on the out-of-core construction of a level-of-detail hierarchy—our framework is general enough to incorporate many different aspects of view-dependent rendering. We therefore emphasize the off-line phases of our method, and report on their theoretical and experimental memory and disk usage and execution time. Our results indicate on average one to two orders of magnitude improvement in processing speed over previous out-of-core methods. Meanwhile, all phases of the method are both disk and memory efficient, and are fairly straightforward to implement.

## 1 INTRODUCTION

Recent advances in scanning technology and the ever increasing size of computer simulations have lead to a rapid increase in the availability and size of geometric data sets. Massive polygonal data sets, consisting of hundreds of millions of faces, are becoming quite common [1, 13]. While the performance of graphics hardware has also seen a dramatic rise in the last few years, our ability to produce data sets that overload the capabilities of state-of-the-art graphics chips has lead researchers to develop methods for automatic model simplification and run-time level-of-detail (LOD) management [7]. Whereas many polygonal environments consist of a large collection of moderately complex objects, such as those used in video games, recent trends are for single objects to consist of millions of polygons. Examples of such models include terrain surfaces and high-resolution range scans. The traditional approach of storing a few static levels of detail is not viable for such large objects, which are often viewed in a manner that they vary greatly in screen resolution over the surface. As a result, methods for view-dependent simplification have been proposed, in which a continuous level-of-detail hierarchy is first constructed, and is then adapted at a fine granularity at run-time [3, 5, 6, 11, 16, 19, 26].

The techniques mentioned above have been used successfully for simplifying models up to a few million triangles. In the last few years, however, polygonal models have become so large that they often greatly exceed our ability to perform conventional in-core simplification, and a number of techniques have been devised for simplifying such large models out-of-core, e.g. [1, 15, 18, 24]. These methods, however, all produce static, single-resolution meshes, which in a sense is inappropriate for such large surfaces, because in order to view them interactively they have to be simplified to such a degree that many important details are lost. Rather, we would like to construct a level-of-detail hierarchy for such large surfaces, and then use view-dependent techniques to render and explore them at varying resolution without significant loss in fidelity. While there have been some recent publications on out-of-core construction of LOD hierarchies, most notably the work by El-Sana and Chiang [4] and by Prince [21], these methods have rather long execution times, can be somewhat difficult to implement, and still rely on having a large amount of memory available for in-core processing. Furthermore, the results presented in these papers are for models of rather modest size—just a million or a few million triangles, making it difficult to judge how well they scale to truly large models.

In this paper, we present an alternative approach to out-of-core simplification for view-dependent refinement, by extending the static simplification algorithm by Lindstrom and Silva [18]. Our end-to-end off-line method is *memory insensitive*, meaning that it can run successfully with essentially an arbitrarily small amount of memory (in our case, less than 8 MB). We achieve this by storing all intermediate computations in temporary files on disk, and take advantage of fast sequential disk access. In addition to being memory efficient, our method is considerably faster than previously published techniques, running at a triangle reduction rate of up to 50,000 triangles per second. We also present data structures for concisely encoding *quadric matrices* and the per-vertex information needed later at run-time during view-dependent rendering. Finally, our method is straightforward to implement, and is a simple but significant extension of Lindstrom and Silva's memory insensitive simplification algorithm. We will present the various steps in our algorithm after covering related work in the area.

## 2 PREVIOUS WORK

In this section, we cover previous work on view-dependent and out-of-core simplification. Because there has been extensive work on simplification, we here only cover a few of the more notable algorithms in the field.

Among the first methods for view-dependent simplification for general polygonal models was the technique by Xia and Varshney [26]. Their method uses *edge collapse* to construct a binary tree of possible coarsening operations off-line. At run-time, they use a screen space metric, based on geometric error and proximity to silhouettes and specular highlights, for determining which edges to collapse. Hoppe extended his work on progressive meshes to view-dependent refinement [11]. Similar to [26], Hoppe uses edge collapse, but his algorithm provides greater freedom in choosing the order of edge collapses, which generally results in higher quality adaptive meshes. His method also makes use of geomorphing to

Contact information: 7000 East Avenue, L-560 · Livermore, CA 94551 · pl@llnl.gov.

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48.

reduce temporal LOD artifacts. More recently, El-Sana and Varshney [5] presented a method, based on the more general vertex merge operation, that is able to merge and simplify topologically disjoint parts of an object.

A different approach to view-dependent refinement was taken by Luebke and Erikson [19]. Rather than relying on edge collapse or vertex pair contraction, they use an even more general coarsening operation—vertex clustering—which allows a large collection of vertices to be merged in a single atomic operation. Like us, they make use of an octree decomposition of space, rather than a general binary tree over the set of mesh vertices. Like others, they make use of normal cones to detect when the surface is near a silhouette. Luebke and Hallen [20] later used this framework for performing view-dependent "imperceptible simplification."

The off-line processing techniques above all assume that the hierarchy can be constructed in-core. To simplify large meshes out-of-core, Lindstrom [15] proposed a technique, based on vertex clustering on a uniform grid, that makes use of Garland and Heckbert's *quadric error metric* [9]. Lindstrom's method performs a single sweep over the mesh, and constructs an in-core representation of the simplified model. More recently Lindstrom and Silva [18] extended this method by removing the requirement of having enough RAM to store the simplified model. Their memory insensitive technique uses a constant amount of memory, and makes use of a series of external sorts to allow sequential access to the on-disk data. Our first simplification phase is based upon their algorithm. To provide a higher level of adaptivity, Shaffer and Garland [24] proposed making two instead of one passes over the mesh. In the first pass, uniform clustering like in [15] is performed, after which a binary space partitioning (BSP) tree is constructed from the accumulated quadric information. In the second pass, the BSP tree is used to recluster the mesh.

There have only been a few published methods for out-of-core simplification for view-dependent refinement. Hoppe applied his view-dependent progressive mesh work to terrain data [12]. His approach is to partition the terrain into a block hierarchy and simplify the blocks independently. Then, the blocks are merged and the seams between them are simplified further. Prince [21] later extended Hoppe's out-of-core simplification method for terrain to arbitrary polygonal surfaces. Like our method, Prince's makes use of quadric error metrics, but uses edge collapse as the coarsening primitive. While effective for medium-size models, his out-of-core method still requires much RAM and may be too slow for simplifying very large models. El-Sana and Chiang [4] proposed a novel out-of-core technique for view-dependent simplification by segmenting the mesh into independent patches. These patches are such that the edge collapse order inherently preserves the boundaries between them, thus simplifying and stitching the patches together can be done without the need for explicit boundary constraints. Unfortunately, the models used in their paper are small by out-of-core simplification standards, and it is not clear how well their method scales.

Rusinkiewicz and Levoy [23] proposed an interesting alternative to polygon-based view-dependent refinement. Their *QSplat* algorithm clusters the triangle mesh into a vertex hierarchy, and then uses point primitives to render the mesh. While conceptually simple, most current hardware is optimized for triangle rather than point rendering, and the quality afforded by real-time point-based rendering can be rather low. Still, hybrid techniques like Cohen et al.'s point- and triangle-based simplification [2] may prove useful.

## 3 ALGORITHM OVERVIEW

Our view-dependent algorithm consists of three phases: simplification, level-of-detail hierarchy construction, and run-time view-dependent refinement and rendering. The first two phases are run off-line, and are used to produce an on-disk level-of-detail representation of the mesh. The run-time component then traverses this hierarchy, pages in the data needed, and produces an adaptive mesh that can be displayed interactively. Our main approach is to use a sparse octree decomposition of space over a uniform rectilinear grid, similar to Luebke and Erikson's view-dependent simplification algorithm [19]. The octree is sparse in the sense that only those nodes that contain at least one vertex from the input mesh are retained. Each node in this octree corresponds to a vertex at some level of resolution, and adaptive mesh simplification and refinement are performed by collapsing and expanding nodes (i.e. removing and creating child nodes, respectively) in the octree. In this section, we give a brief overview of the three phases of our algorithm, and provide further details in the following sections.

The first phase—simplification—is based on the *OoCSx* memory insensitive mesh simplification algorithm by Lindstrom and Silva [18]. Their method is a memory efficient variation on the out-of-core simplification algorithm by Lindstrom [15], which in turn was inspired by Rossignac and Borrel's [22] vertex clustering algorithm. Using a uniform grid to partition space, all vertices that fall in the same grid cell are merged (clustered) to a single vertex. In this process, the triangles that collapse to an edge or a point are discarded. Representative vertices for the clusters are chosen based on minimizing the *quadric error*—a weighted sum of squared distances to the triangle planes of the input mesh—which can be encoded using a symmetric $4 \times 4$ matrix [9].

The OoCSx algorithm performs all these tasks on disk, and avoids costly random accesses using a sequence of external sorts, followed by fast sequential accesses. The intermediate output of this algorithm is a set of triangles for the simplified mesh, and a list of quadric matrices for its vertices. We associate with each quadric matrix an *octcode*—a bit string that uniquely identifies a grid cell by position and resolution in the octree. Furthermore, the quadric file is output in octcode order, such that sibling nodes are stored together. The triangle and quadric files constitute a complete representation of the simplified mesh, and this is the only data output by our simplification phase. This phase of the algorithm is described in Section 4.

The LOD construction phase begins by processing the quadric matrix file sequentially, grouping sibling nodes together, and producing a new quadric matrix for the parent node as the sum of the children's quadric matrices. The parent nodes are then output sequentially (again in octcode order) to a temporary file for that given level of resolution. This process is then applied iteratively until all levels in the octree are represented, ending with a temporary file containing a single node—the root node. This bottom-up construction is then followed by a top-down, level-by-level traversal of the quadric files, during which optimal vertex positions and quadric errors are computed and written to the final output file. The triangle file, which is treated as a first-in-first-out queue, is simultaneously traversed sequentially, and the triangles are distributed among the current node and its children. A triangle is assigned to a node if it degenerates when the node is collapsed (cf. [19]). Eventually all the nodes of the octree have been output, and all the triangles have been assigned to the internal nodes of the octree. The algorithm for the LOD construction phase is described in Section 5, while the data structures for the output produced are discussed in detail in Section 6.

In the run-time phase, the file containing the LOD hierarchy, which could potentially greatly exceed the available main memory, is memory mapped so that it can be accessed as though it were resident in contiguous memory. For simplicity, we let the operating system perform on-demand paging of the external data. At the beginning of each frame, we perform view-dependent refinement by accessing a dynamically allocated in-core "copy" of the currently active nodes in the static LOD hierarchy. This refinement

```
simplify($T_{in}$)
for each triangle $t = \langle \mathbf{p}_1^t, \mathbf{p}_2^t, \mathbf{p}_3^t \rangle \in T_{in}$
    compute plane equation $\bar{\mathbf{n}}_t$ for $t$
    for each vertex $\mathbf{p}_i^t$ of $t$
        map $\mathbf{p}_i^t$ to leaf octcode $v_i^t$
        append $\langle v_i^t, \bar{\mathbf{n}}_t \rangle$ to plane equation file $P$
    if $v_1^t, v_2^t, v_3^t$ are distinct then
        append $\langle v_1^t, v_2^t, v_3^t \rangle$ to triangle file $T_{out}$
externally sort $P$ on octcode $v$
for each octcode $v \in P$
    for each plane equation $\bar{\mathbf{n}}_t$ for $v$
        add $\bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^{\mathsf{T}}$ to quadric matrix $\mathbf{Q}_v$ for $v$
    append $\langle v, \mathbf{Q}_v \rangle$ to quadric file $Q_n$
```

Table 1: Pseudo-code for memory insensitive simplification. The output is a triangle file $T_{out}$ and a quadric file $Q_n$ for level $n$ (the bottom level) in the LOD hierarchy.

is performed recursively in a top-down, depth-first traversal. If the projected quadric error for a node exceeds a user-specified threshold, then we expand the node. Conversely, if the error is smaller than the threshold, then we collapse the node. As new nodes are created, we extract data from the external LOD hierarchy, compute any per-vertex and per-triangle information not explicitly stored, and write this data to a dynamically allocated data structure. Finally, the nodes in this adaptive octree are visited (in no particular order), and all triangles encountered are rendered. The in-core, dynamic data structures are presented in Section 7, while the steps of the view-dependent refinement are given in Section 8.

# 4 SIMPLIFICATION

The simplification phase of our algorithm is based on, and is essentially identical to, the beginning stages of the memory insensitive technique *OoCSx* described in [18]. Therefore, we will only briefly cover this part of our algorithm, and we will focus on the few differences between the two methods. Pseudo-code for the simplification phase is given in Table 1.

Before the simplification begins, the user chooses the resolution of a uniform rectilinear grid that completely contains the input mesh. This grid is constrained to have dimensions $2^n \times 2^n \times 2^n$, for some positive integer $n$. The cells in this grid correspond to the leaf nodes in an $(n+1)$-level octree, that ultimately forms a multiresolution representation of the mesh. In the discussion below, we will use the terms grid cell, cluster, and node interchangedly.

As in [18], we process the input mesh, which is represented as a triangle soup (i.e. a sequence of triplets of vertex coordinates), one triangle at a time. For each triangle $t$, we compute a 4-vector

$$\bar{\mathbf{n}}_t = A_t \begin{pmatrix} \hat{\mathbf{n}}_t \\ d_t \end{pmatrix} \qquad (1)$$

for an implicit plane equation $\hat{\mathbf{n}}_t^{\mathsf{T}} \mathbf{x} + d_t = 0$. Note that $\bar{\mathbf{n}}_t$ is weighted by the area $A_t$ of $t$. Then, for each of $t$'s three vertices, we quantize the vertex coordinates to an integer grid cell location, and then convert this location to an *octcode* $v$. These octcodes are represented as follows: The root node has octcode $v = 1$, and the $k^{th}$ child of a node $v$ is computed as $8v + k$. These octcodes have the property that they are ordered by level, from top to bottom, and sibling nodes have consecutive octcodes. Whereas these octcodes are different from the cluster IDs used in OoCSx, this is of no consequence to the simplification algorithm—any one-to-one mapping between quantized coordinates and cluster IDs will do. The plane equations and associated octcodes are then output sequentially to a temporary plane equation file, $P$, and non-degenerate triangles are written to a triangle file $T_{out}$.

After the input has been exhausted, the plane equation file is sorted on the octcode field using an external sort. As in [18], we use `rsort` [14] for this task. We then process all plane equations, one cluster at a time, and construct a 4 × 4 quadric matrix $\mathbf{Q}_v$ for each cluster $v$:

$$\mathbf{Q}_v = \sum_t \bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^{\mathsf{T}} \qquad (2)$$

```
octree-construct($Q_n$, $F$)
for each level $l = n, ..., 1$
    for each set of siblings $C$ in $Q_l$ with parent octcode $p$
        compute parent quadric $\mathbf{Q}_p = \sum_{c \in C} \mathbf{Q}_c$
        append $\langle p, \mathbf{Q}_p \rangle$ to parent quadric file $Q_{l-1}$
for each level $l = 0, ..., n$
    for each node $\langle p, \mathbf{Q}_p \rangle \in Q_l$
        add link to $p$ from its parent in $H$
        compute vertex data from $\mathbf{Q}_p$ and append to $H$
        if $head(F) = p$ then
            dequeue triangles $T_p$ from $F$
            for each triangle $t = \langle v_1^t, v_2^t, v_3^t \rangle \in T_p$
                if $\{path(v_i^t, l+1)\}$ are distinct then
                    append $t$ to current node $p$ in $H$
                else
                    identify common child $c$ from $\{v_i^t\}$
                    append $t$ to temporary file $T_c$
            for each child $c$ of $p$
                if $T_c$ is non-empty then
                    enqueue $\langle c, T_c \rangle$ onto $F$
```

Table 2: Pseudo-code for memory insensitive LOD hierarchy construction. This procedure takes as input a quadric file $Q_n$ and a queue $F$ of triangles to process for the root node, and outputs an LOD hierarchy $H$.

Finally, we output the quadric matrices along with their octcodes to a quadric file $Q_n$ for level $n$ in the octree (the bottommost level), and we remove the temporary plane equation file $P$. For each leaf node that contains at least one vertex from the original mesh, we have a quadric matrix that corresponds to exactly one vertex in the simplified mesh. The original OoCSx simplification algorithm would at this point compute representative vertices for each cluster. For storage efficiency reasons, we will defer this computation until later. OoCSx would then proceed by performing multiple external sorts on the triangle file to replace the cluster IDs with vertex indices. Because we will make direct use of the cluster IDs (or octcodes), this step fortunately does not have to be done in our algorithm. Instead, we perform only a single external sort—on the plane equation file.

There is one important detail that we have left out so far, and which we will revisit later. During the LOD hierarchy construction, we will need an approximate surface normal for each node. When adding up the plane equations during simplification, we can easily construct such normals for each node in $Q_n$. In our implementation, we store these normals together with the quadric matrices in $Q_n$, and later propagate them up the octree. If space is at a premium, then an alternative approach would be to extract the normals from the quadric matrices. We will show in Section 6.1.2 how to do this.

The steps described above are all the components of our simplification algorithm. The output is a single-resolution, static mesh, represented as $\langle Q_n, T_{out} \rangle$. We now proceed by constructing a level-of-detail hierarchy for this simplified mesh.

# 5 LOD HIERARCHY CONSTRUCTION

The second phase of our algorithm takes the simplified mesh, represented as a list of per-vertex quadric matrices $Q_n$ and a list of triangles $T_{out}$, and constructs a coarse-to-fine level-of-detail representation $H$ of the mesh. For each node in $H$, we store vertex information, such as position, normal, error, etc., as well as a (potentially empty) list of triangles. These triangles are the ones that are eliminated when the node is collapsed. We will focus later on the particular data structures used for the nodes in $H$, and spend this section describing the steps of the LOD construction algorithm.

Table 2 lists pseudo-code for the octree construction. We begin by computing quadric matrices for the interior nodes of the octree (lines 1–4). Recall that the simplification has already produced quadric matrices $Q_n$ for the leaf nodes on level $n$. Because $Q_n$ is sorted on octcode, and because sibling nodes have consecutive octcodes, we can easily scan $Q_n$ and fetch the quadric matrices for each group of siblings. From these we compute a quadric matrix for the parent using simple matrix addition. The resulting matrix is then output to another temporary file for the next coarser level. This procedure is iterated until quadric matrices for all levels of the octree have been constructed.

The next and final step is to compute vertex data, assign triangles to each node, and create links from each parent to its children. As is common in multiresolution methods, we store the multiresolution structure from coarse to fine resolution. While this layout is of no particular advantage to our view-dependent algorithm, other than the fact that the breadth-first layout and closeness of siblings in the file result in good cache coherence, we anticipate that a coarse-to-fine order would be beneficial for progressive transmission. In addition, constructing $H$ in this order allows the triangles to be quickly distributed to their respective nodes.

We proceed by initializing a first-in-first-out (FIFO) queue $F$ with the tuple $\langle r, T_{out}\rangle$, where $r = 1$ is the root octcode and $T_{out}$ is the entire set of triangles in the simplified mesh. Then, for each level $l$ starting at the root, we read nodes sequentially from the quadric file $Q_l$ on level $l$. When processing a new node $p$, we first identify its parent (line 7). Due to our octcode construction, either $p$ has the same parent $q$ as the previously processed node, or $p$'s parent is the node immediately following $q$. By maintaining the octcode $q$ for the current parent, we test if $q$ is $p$'s parent. If not, we recompute $q$ from $p$ and move on to the next node in $H$ (which is guaranteed to be $q$). We then add a link (file offset) from $q$ to its child $p$. Note that adding child links requires both read and write random access to $H$, but these accesses are coherent and do not incur excessive overhead.

From the quadric matrix $\mathbf{Q}_p$ for the current node $p$, we compute the per-vertex data (see Section 6) needed at run-time, and append this data to the output file $H$. If the node at the head of the queue $F$ equals the current node $p$, then there are triangles $T_p$ to be processed for $p$. (Note that this condition always fails for leaf nodes, which do not contain triangles.) For each triangle $t$, we examine its three octcodes $\langle v_1^t, v_2^t, v_3^t\rangle$ to determine if it belongs to the current node or to one of its children. Note that each group of three bits in an octcode determine which of eight branches to take from the node arrived to so far from the root. Thus, by examining the paths taken from the root to one level below $p$ (i.e. up to level $l+1$; see line 12), and testing if these paths are all different for $\{v_i^t\}$, we can determine if expanding $p$ would cause the previously degenerate triangle to become non-degenerate. If this is the case, then we add $t$ to the current node $p$ being output. Otherwise, it must be the case that at least two of the paths coincide. Furthermore, these coincident paths must lead to one of the children $c$ of $p$, and we append the triangle to a temporary triangle file $T_c$ for that child.

After all the triangles $T_p$ have been processed, we enqueue (append) all the non-empty temporary triangle files $T_c$ (in octcode order) onto the FIFO $F$,[1] and we are then done processing node $p$. After the temporary file $Q_l$ has been exhausted, we can remove it and move on to the next level. Finally, after all nodes have been output to $H$, we remove $F$, and phase 2 of our algorithm is complete.

| | | |
|---|---|---|
| external octree node | | |
| vertex data | | |
| **3-vector** | $\mathbf{r}$ | rotation for orthogonal matrix $\mathbf{P}$ |
| **3-vector** | $\boldsymbol{\lambda}$ | eigenvalues of $\mathbf{A}$ |
| **3-vector** | $\mathbf{p}$ | vertex position |
| **scalar** | $\epsilon$ | quadric error |
| triangle data | | |
| **count** | $n_T$ | number of triangles |
| **octcode*3** | $T$ | list of triangles |
| octree data | | |
| **offset*8** | $c$ | file offsets to children |

Table 3: External data structures for internal (i.e. non-leaf) octree node. Leaf nodes contain only the vertex data fields.

Note that the size of $F$ is linear in the number of triangles $T_{out}$ and the number of nodes. Because the number of triangles in $F$ is limited by $T_{out}$, and since we already know the total number of nodes, it is possible to use a fixed amount of disk space for $F$ by using circular storage. However, for simplicity and because $F$ is generally small compared to the overall disk usage we did not use such a circular queue in our implementation.

## 6 EXTERNAL DATA STRUCTURES

We now turn our attention to the data structures used for our external on-disk representation of the level-of-detail hierarchy $H$. Each node in the hierarchy consists of vertex information (position, error, etc.) and, if the node is not a leaf, a list of triangles and pointers to its children. These data structures are given by Table 3. The triangles are represented as triplets of octcodes corresponding to leaf nodes in the hierarchy. The computation of the vertex data from the quadric matrices is more involved, and we will describe the vertex fields stored in the following sections.

### 6.1 Vertex Data

In this section, we describe how to compute and store all the per-vertex information needed in our view-dependent renderer from the $4 \times 4$ quadric matrix $\mathbf{Q}$. The data we are concerned with are the vertex position $\mathbf{p}$, the surface normal $\hat{\mathbf{n}}$, the quadric error $\epsilon$ at $\mathbf{p}$, and a matrix $\mathbf{K}$ that encodes the normal curvature and is used to measure how large the error appears from different view directions. We could compute and store $(\mathbf{K}, \hat{\mathbf{n}}, \mathbf{p}, \epsilon)$ directly, however this information would require $6+3+3+1 = 13$ scalar values, whereas the original quadric matrix requires only 10 values (assuming we take advantage of the fact that $\mathbf{Q}$ and $\mathbf{K}$ are symmetric). Instead, we will make use of an alternative representation $(\mathbf{r}, \boldsymbol{\lambda}, \mathbf{p}, \epsilon)$—three 3-vectors and a scalar—that allows $\mathbf{K}$ and $\hat{\mathbf{n}}$ to be computed quickly.

#### 6.1.1 Vertex Position and Quadric Error

In Section 4, we explained how to compute the quadric matrix $\mathbf{Q}$ for a vertex or, more generally, a node in the LOD hierarchy. As in [15], we decompose the quadric matrix as

$$\mathbf{Q} = \begin{pmatrix} \mathbf{A} & -\mathbf{b} \\ -\mathbf{b}^\mathsf{T} & c \end{pmatrix} \tag{3}$$

We can then write the quadric error $Q$ as

$$\begin{aligned} Q(\mathbf{x}) &= \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} - 2\mathbf{b}^\mathsf{T}\mathbf{x} + c \\ &= (\mathbf{x}-\mathbf{p})^\mathsf{T}\mathbf{A}(\mathbf{x}-\mathbf{p}) \\ &\quad - 2\mathbf{b}^\mathsf{T}(\mathbf{I}-\mathbf{A}^+\mathbf{A})\mathbf{x} + (c - \mathbf{b}^\mathsf{T}\mathbf{A}^+\mathbf{b}) \end{aligned}$$

where $\mathbf{A}^+$ is the pseudo-inverse [10] of $\mathbf{A}$, and $\mathbf{p}$ is a point at which the quadric error is minimized. If $\mathbf{A}$ is singular, then $Q$ has infinitely many minima. If however $\mathbf{A}$ is non-singular, which

[1] This enqueuing operation could be implemented by simply adding a pointer to each temporary file $T_c$. However, this would eventually lead to a very large number of files (possibly millions), and could result in a significant space and time overhead when locating and opening the files.

happens in the vast majority of cases, then $\mathbf{A}^+ = \mathbf{A}^{-1}$, and the quadric error reduces to

$$\begin{aligned} Q(\mathbf{x}) &= (\mathbf{x}-\mathbf{p})^{\mathsf{T}}\mathbf{A}(\mathbf{x}-\mathbf{p}) + (c - \mathbf{b}^{\mathsf{T}}\mathbf{A}^{-1}\mathbf{b}) \\ &= (\mathbf{x}-\mathbf{p})^{\mathsf{T}}\mathbf{A}(\mathbf{x}-\mathbf{p}) + \epsilon \end{aligned} \tag{4}$$

where $\epsilon = Q(\mathbf{p})$ is the minimum quadric error. That is, we can parameterize the quadric error as $(\mathbf{A}, \mathbf{p}, \epsilon)$. Note that $\mathbf{p}$ is generally our chosen vertex position. The only exception is when $\mathbf{p}$ falls outside its associated grid cell, in which case we constrain the position using the procedure outlined in [18].

Regardless of the rare special cases mentioned above, if we do not have to compute $Q$ at positions other than the vertex position $\mathbf{p}$ (irrespective of our choice of $\mathbf{p}$), then the parameterization $(\mathbf{A}, \mathbf{p}, \epsilon)$ is useful, since it directly gives us the error $\epsilon$ and vertex position $\mathbf{p}$. Still, we are left with determining the normal $\hat{\mathbf{n}}$ and the curvature matrix $\mathbf{K}$. As shown below, these two quantities can both be derived from the matrix $\mathbf{A}$.

### 6.1.2 Surface Normal Encoding

To compute the surface normal $\hat{\mathbf{n}}$, note that the matrix $\mathbf{A}$ is the covariance matrix (with zero mean) for the set of (weighted) normals of the triangles in the cluster [8]. Thus, the eigenvector for the largest eigenvalue $\lambda_1$ of $\mathbf{A}$ corresponds to the dominant normal direction $\hat{\mathbf{n}}$. Note that if $\hat{\mathbf{n}}$ is an eigenvector, then so is $-\hat{\mathbf{n}}$. Because the sign of the normal matters for correct rendering, we will show later how to resolve this ambiguity. Using an eigen decomposition of $\mathbf{A}$, we have

$$\mathbf{A} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^{\mathsf{T}} \tag{5}$$

$$\mathbf{P} = (\hat{\mathbf{n}} \quad \hat{\mathbf{x}}_2 \quad \hat{\mathbf{x}}_3) \tag{6}$$

$$\mathbf{\Lambda} = \mathrm{diag}(\lambda_1, \lambda_2, \lambda_3) \quad \lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0 \tag{7}$$

where $\mathbf{\Lambda}$ is a diagonal matrix of (non-negative) eigenvalues, and $\mathbf{P}$ is orthogonal with determinant $\det(\mathbf{P}) = 1$. That is, $\mathbf{P}$ is a rotation matrix, which can be represented using as little as three parameters. We have chosen to use a 3-parameter axis-angle representation that is similar to the standard unit quaternion representation. Let $\mathbf{P}$ correspond to a rotation around a unit vector $\hat{\mathbf{r}}$ by an angle $\theta$. Then the vector

$$\mathbf{r} = (r_x \quad r_y \quad r_z)^{\mathsf{T}} = \sqrt{2}\sin\tfrac{\theta}{2}\,\hat{\mathbf{r}} \tag{8}$$

completely represents $\mathbf{P}$. We will not go into the details of how to compute $\mathbf{r}$ from $\mathbf{P}$, but refer the reader to any tutorial on quaternions, e.g. [25]. We recover $\mathbf{P}$ from $\mathbf{r}$ as follows:

$$\mathbf{P} = \begin{pmatrix} 1 - r_y^2 - r_z^2 & r_x r_y - \alpha r_z & r_x r_z + \alpha r_y \\ r_y r_x + \alpha r_z & 1 - r_z^2 - r_x^2 & r_y r_z - \alpha r_x \\ r_z r_x - \alpha r_y & r_z r_y + \alpha r_x & 1 - r_x^2 - r_y^2 \end{pmatrix} \tag{9}$$

$$\alpha = \sqrt{2 - r_x^2 - r_y^2 - r_z^2} \tag{10}$$

Thus, by storing $\mathbf{r}$, we can quickly compute $\mathbf{P}$ and the normal $\hat{\mathbf{n}}$ from the first column of $\mathbf{P}$. To recover $\mathbf{A}$, we also store the eigenvalues $\boldsymbol{\lambda}$.

As noted above, the canonical decomposition $\mathbf{A} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^{\mathsf{T}}$ does not necessarily lead to a matrix $\mathbf{P}$ whose first column equals the surface normal in sign. For example, $\mathbf{A} = (-\mathbf{P})\mathbf{\Lambda}(-\mathbf{P})^{\mathsf{T}}$ is an equally valid decomposition. However, because this computation is deterministic, we will always obtain the same matrix $\mathbf{P}$ from any given $\mathbf{A}$. Thus, if we already know the (approximate) normal, which is the case in our simplification algorithm (see Section 4), then we can test whether the normal obtained from $\mathbf{P}$ matches the given normal. If the two vectors point in opposite directions, then we encode this fact by negating $\lambda_1$. Because $\mathbf{A}$ is non-zero and positive semi-definite, we must have $\lambda_1 > 0$, and we can therefore safely use the sign bit of $\lambda_1$ to encode the sign of $\hat{\mathbf{n}}$.

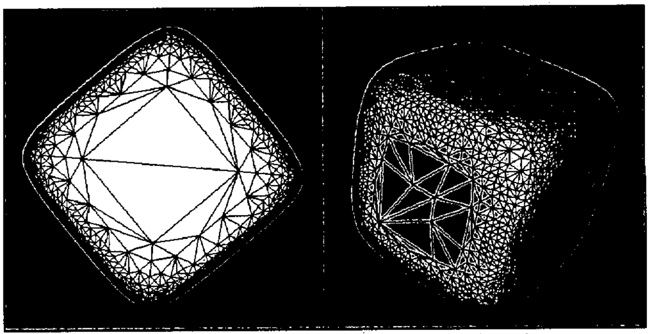

(a) Orthogonal view. (b) Oblique view.

Figure 1: Illustration of silhouette preservation. The nearly flat faces of the cube can be simplified greatly when they are orthogonal to the view direction.

As mentioned in Section 4, we explicitly store approximate normals in the quadric files. To save disk space, however, we could use the technique just described for extracting normals from quadric matrices. We would then compute—but not store—a surface normal for each leaf node during simplification, compare it against the normal obtained from the quadric matrix $\mathbf{Q}$, and encode its sign difference in $\mathbf{Q}$. Similar to the argument above, because $\mathbf{Q}$ is non-zero and positive semi-definite, $\mathrm{tr}(\mathbf{Q}) > 0$, and the need to flip the extracted normal can be encoded by negating the diagonal of $\mathbf{Q}$.

### 6.1.3 Curvature Matrix

Our view-dependent error metric takes advantage of the fact that geometric displacements parallel to the view direction are less perceptible than those orthogonal to the view direction. Thus geometry viewed straight-on can often be coarsened significantly more than geometry near silhouettes—a fact that has been exploited by other view-dependent methods, e.g. [11, 16, 19, 26]. This is illustrated in Figure 1 for a smoothed and slightly curved cube. Rather than using a cone to bound the normals [11, 19], we account for this directionality by analyzing the normal spread given by the quadric matrix.

Let $\hat{\mathbf{n}}_t$ be the unit normal, and consequently the direction of geometric error, associated with triangle $t$. Furthermore, let $\hat{\mathbf{v}}$ be the unit vector from the cluster's representative vertex that we are computing the error for to the viewpoint. In our anisotropic error projection, we modulate the error associated with $t$ by the sine of the angle $\gamma_t$ between $\hat{\mathbf{n}}_t$ and $\hat{\mathbf{v}}$, i.e. by the factor $\eta_t = |\sin\gamma_t| = \|\hat{\mathbf{n}}_t \times \hat{\mathbf{v}}\|$. Thus, when $\gamma_t$ is zero, the projected error vanishes, while $\gamma_t = 90°$ implies that we are near a silhouette, and the projected error is at a maximum. We can rewrite (the square of) $\eta_t$ as follows:

$$\eta_t^2 = \|\hat{\mathbf{n}}_t \times \hat{\mathbf{v}}\|^2 = \hat{\mathbf{v}}^{\mathsf{T}}\hat{\mathbf{v}}\hat{\mathbf{n}}_t^{\mathsf{T}}\hat{\mathbf{n}}_t - (\hat{\mathbf{v}}^{\mathsf{T}}\hat{\mathbf{n}}_t)^2 = \hat{\mathbf{v}}^{\mathsf{T}}(\mathbf{I} - \hat{\mathbf{n}}_t\hat{\mathbf{n}}_t^{\mathsf{T}})\hat{\mathbf{v}} \tag{11}$$

This modulation factor for a single triangle $t$ can then be extended to a set of triangles $T$ in a cluster as a weighted sum:

$$\begin{aligned} \eta_T^2 &= \frac{\sum_{t\in T} A_t^2 \eta_t^2}{\sum_{t\in T} A_t^2} \\ &= \hat{\mathbf{v}}^{\mathsf{T}} \frac{\sum_{t\in T} A_t^2 \mathbf{I} - (A_t\hat{\mathbf{n}}_t)(A_t\hat{\mathbf{n}}_t)^{\mathsf{T}}}{\sum_{t\in T} A_t^2} \hat{\mathbf{v}} \\ &= \hat{\mathbf{v}}^{\mathsf{T}} \left(\mathbf{I} - \frac{\mathbf{A}}{\mathrm{tr}(\mathbf{A})}\right) \hat{\mathbf{v}} \\ &= \hat{\mathbf{v}}^{\mathsf{T}} \left(\mathbf{I} - \frac{\mathbf{P}\mathbf{\Lambda}\mathbf{P}^{\mathsf{T}}}{\lambda_1 + \lambda_2 + \lambda_3}\right) \hat{\mathbf{v}} \\ &= \hat{\mathbf{v}}^{\mathsf{T}}\mathbf{K}\hat{\mathbf{v}} \end{aligned} \tag{12}$$

As in Section 4, $A_t$ is the area of $t$. The expression for the trace of $\mathbf{A}$, $\mathrm{tr}(\mathbf{A}) = \sum_{t\in T} A_t^2$, follows from the fact that $\mathrm{tr}(\cdot)$ is linear and $\mathrm{tr}(\hat{\mathbf{n}}_t\hat{\mathbf{n}}_t^{\mathsf{T}}) = \hat{\mathbf{n}}_t^{\mathsf{T}}\hat{\mathbf{n}}_t = 1$.

| in-core octree node | | |
|---|---|---|
| vertex data | | |
| **3×3-matrix** | $\mathbf{K}$ | curvature matrix |
| **3-vector** | $\hat{\mathbf{n}}$ | vertex normal |
| **3-vector** | $\mathbf{p}$ | vertex position |
| **scalar** | $\epsilon$ | quadric error |
| triangle data | | |
| **count** | $n_T$ | number of triangles |
| **vertex***3 | $T$ | list of triangles |
| **count** | $n_R$ | number of references |
| **pointer** | $R$ | list of references to this node |
| octree data | | |
| **boolean** | *leaf* | is node a leaf in the dynamic octree? |
| **index** | $l$ | octree level |
| **pointer** | $s$ | pointer to static external node |
| **pointer***8 | $c$ | pointers to children |

Table 4: In-core data structures for octree node.

We call $\mathbf{K}$ the "curvature matrix," because it encodes the amount that the surface curves in different directions. From the definition of $\mathbf{K}$, we see that as the normals spread out the curvature grows, the eigenvalues approach a common value, $\mathbf{K}$ approaches $\frac{2}{3}\mathbf{I}$, and as a result $\eta_T$ becomes more and more isotropic. If on the other hand $\lambda_1$ is much larger than the other eigenvalues, then the normals are tightly clustered, the curvature is small, and $\eta_T$ allows aggressive coarsening when $\hat{\mathbf{v}}$ is near the dominant normal direction.

# 7 IN-CORE DATA STRUCTURES

In this section we describe the in-core data structures used in our run-time view-dependent renderer. These data structures closely resemble the ones used by Luebke and Erikson [19], but have been modified to work in an out-of-core setting, where only part of the octree is assumed to be memory resident. That is, we maintain an in-core subset $H'$ of the LOD hierarchy $H$, such that the leaf nodes of $H'$ correspond to the vertices of the adaptively refined mesh. We refer to the subset $H'$ of $H$ as the set of *active* nodes. The triangles of the refined mesh are those stored in the internal (non-leaf) nodes of $H'$. Each triangle vertex, represented on disk as an octcode $v$ for a leaf node in $H$, is mapped to a *proxy* vertex—either $v$ itself or its lowest active ancestor. Thus, the actual vertices used for a triangle change dynamically as the mesh is adaptively refined and simplified.

Table 4 lists the octree node data structures that are dynamically allocated at run-time. Note that these data structures are the minimum information needed to drive our view-dependent refinement, and they may need to be augmented for additional capabilities such as view frustum, occlusion, and back-face culling, more complicated view-dependent error metrics, texturing, etc. We have already covered the fields for the vertex data in Section 6, and will here discuss the remaining fields for the node. The triangle data consists of a list of triangles and a list of pointers to triangle vertices that currently reference this node as their proxy. Thus the **vertex** data type consists of an octcode for a leaf node and an octree node pointer to the vertex's proxy. Whenever a node is expanded, we need to modify all the triangle vertices that have the node as a proxy, which is accomplished by maintaining a list of back references to those vertices. Similarly, when a node is collapsed, its children's references are first accessed, and the associated proxies are modified to point to the collapsed node.

In contrast to [19], where the entire hierarchy is assumed to be memory resident, we cannot pre-compute the list of back references (or "tris" using their terminology), because these references might point to triangles in nodes that are not active and therefore have not been paged in. Rather, as triangles are added and removed, we add and remove references on-the-fly as necessary. Similarly, we do not construct the triangle list $T$ for a node until it is expanded. We will further discuss the operations on the octree $H'$ in the following section.

```
node-expand($p$)
  $leaf(p) \leftarrow$ false
  for each child $c$ of $p$
      initialize $c$ by computing its vertex data
  for each triangle $t$ assigned to $p$
      for each vertex $v_i^t$ of $t$
          identify proxy node $q_i^t$ from octcode $v_i^t$
          add reference $r$ from proxy node $q_i^t$ to $t$
  for each reference $r$ in $p$
      identify child $c$ corresponding to $r$
      transfer reference $r$ from $p$ to $c$

node-collapse($p$)
  for each child $c$ of $p$
      node-collapse($c$)
      transfer reference $r$ from $c$ to $p$
      set $r$'s proxy to point to $p$
      remove $c$
  for each triangle $t$ assigned to $p$
      for each vertex $v_i^t$ of $t$
          remove reference to $t$ from proxy $q_i^t$ for $v_i^t$
  $leaf(p) \leftarrow$ true
```

Table 5: Pseudo-code for node expansion and collapse.

# 8 VIEW-DEPENDENT REFINEMENT

We are now ready to describe the steps pertaining to the final phase of our out-of-core view-dependent renderer; the run-time component. We reiterate that the view-dependent refinement algorithm presented here is in a sense minimalistic, and serves mostly as a proof of concept for showing the correctness of our memory insensitive simplification, that our data structures are appropriate, and that our run-time framework and interface with the out-of-core data are efficient enough to achieve interactive frame rates. It is possible to improve the efficiency of our run-time system, for example by incorporating support for various types of culling, including view frustum, back-face, and occlusion culling. However, such techniques are beyond the scope of this paper, and we see them as fruitful avenues for future work. We here describe our run-time framework, with the main focus on its out-of-core aspects.

Because the level-of-detail hierarchy stored on disk may exceed the amount of available memory, we must be careful to page in only the active nodes of the hierarchy. Rather than making use of an explicit paging system, we rely on the use of read-only *memory mapping* to associate the on-disk hierarchy with a logically contiguous address space, and let the operating system fetch the data from disk when it is first accessed. Similar strategies for out-of-core rendering of large terrain have been employed, for example by Hoppe [12] and by Lindstrom and Pascucci [17]. This approach to data paging is particularly attractive when the refinement and rendering tasks are decoupled and run asynchronously. Also, as demonstrated in [17], by arranging the data and the accesses to it in a cache coherent manner, it is possible to substantially improve the paging performance. Indeed, our choice of arranging the octree in a coarse-to-fine, breadth-first layout on disk was made intentionally, after having been inspired by the quadtree layout in [17]. Given this general framework, we now describe the two tasks of adaptive refinement and rendering.

## 8.1 Refinement Algorithm

Similar to several other components of our algorithm, our adaptive octree refinement closely follows the strategy employed by Luebke and Erikson [19]. We begin by creating a single node for the root of the dynamic octree $H'$. During refinement, we make use of two complementary operations; node expansion and collapse. When expanding a node we add its children; when collapsing a node we remove its descendants. Pseudo-code for these steps is listed in Table 5.

At the beginning of each frame, we recursively visit the nodes in the octree, from top to bottom, and evaluate the refinement criterion for each node by projecting its quadric error onto the screen. (The details of this evaluation are given below.) If the error exceeds a user-specified threshold $\tau$, then we continue the depth-first traversal. If we reach a leaf node in $H'$ that needs to be refined, then we expand this node and visit its children. If at any point the projected error is smaller than the threshold, then we collapse the node by discarding all of its descendants. In this manner, the octree adapts as the viewpoint changes, and we visit only those nodes that eventually make up the mesh.

Finally, after the octree has been refined, we traverse it node by node and render all the triangles encountered. For each triangle, we follow the pointers to the proxy nodes, and send their vertex positions and normals to the rendering subsystem.

### 8.2 Screen Space Metric

The decision whether to collapse or expand a node is governed by a screen space error metric and a user-specified error threshold. In our screen space metric, we make use of the quadric error $\epsilon$ and position $\mathbf{p}$ of a node's representative vertex, as well as the curvature matrix $\mathbf{K}$. Note that, due to our triangle-area-weighting of the geometric displacements (see Section 4), our quadric error has units of volume squared, which needs to be expressed in units of length. This can be done, for example, by normalizing the error by dividing by the sum of squared areas for the cluster's triangles (i.e. by the sum of eigenvalues $\lambda_1 + \lambda_2 + \lambda_3$). Another approach, and the one used in our implementation, which ensures that the error of a node is at least as large as its children's, is to assume that the volumetric errors correspond to some hypothetical volume, e.g. a sphere, in which case we simply compute the radius of the sphere and use it as the error term. In either case, these computations are done once when the node is constructed, and the $\epsilon$ field in the dynamic node is assumed to have units of length.

As a base metric, we set the screen space error to be proportional to the ratio of the object space error and the distance from the viewpoint to the node. As suggested in Section 6.1.3, to incorporate directionality into our metric for silhouette preservation, we modulate the base metric by the factor $\eta$. Thus, our screen space metric $\rho$ can be written as

$$\rho = \lambda\eta\frac{\epsilon}{\|\mathbf{e}-\mathbf{p}\|} = \lambda\epsilon\frac{\sqrt{\mathbf{v}^\mathsf{T}\mathbf{K}\mathbf{v}}}{\mathbf{v}^\mathsf{T}\mathbf{v}} \tag{13}$$

where $\mathbf{v} = \mathbf{e}-\mathbf{p}$ is the vector from the node to the viewpoint $\mathbf{e}$, and $\lambda$ is the screen resolution in pixels per radians. We then compare $\rho$ against a threshold $\tau$ to determine whether the node is active or not. For efficiency reasons, we square and rearrange some terms, and obtain the following expression:

$$\begin{aligned} \mathit{active} &\iff \rho > \tau \\ &\iff \lambda^2\epsilon^2(\mathbf{v}^\mathsf{T}\mathbf{K}\mathbf{v}) > \tau^2(\mathbf{v}^\mathsf{T}\mathbf{v})^2 \\ &\iff \epsilon^2(\mathbf{v}^\mathsf{T}\mathbf{K}\mathbf{v}) > \kappa^2(\mathbf{v}^\mathsf{T}\mathbf{v})^2 \end{aligned} \tag{14}$$

where $\kappa = \frac{\tau}{\lambda}$ is the screen space error threshold in radians.

Because the octree is pruned whenever a node is found to be inactive, we should ideally ensure that a node's projected error is always larger than those of its children. Note that the quadric errors by their additive nature already satisfy this nesting property. Similarly, the amount of curvature encoded in the matrix $\mathbf{K}$ can only increase when quadric matrices are combined. However, the direction and length of the vector $\mathbf{v}$ varies from node to node, making it possible to violate the nesting condition. A general technique for handling this view-dependent problem was presented in [17], in which a nested sphere hierarchy is computed and used in place of the positions of individual vertices. While not implemented here, all the information needed for constructing this hierarchy is readily available in our off-line simplification algorithm, and we believe that it would be rather straightforward to incorporate this sphere hierarchy into our screen space metric.

## 9 RESULTS

In this section we present experimental results of running our algorithms. We used a number of polygonal test models, including a massive 373 million triangle model of Michelangelo's St. Matthew statue [13]. All models, except the Buddha, were simplified on an SGI Onyx2 with forty-eight 250 MHz R10000 CPUs and 15.5 GB of main memory. The Buddha model was simplified and later rendered on a Linux PC with two 800 MHz Pentium III processors, 512 MB of RAM, and a GeForce3 graphics card. As is evident in the accompanying video, which shows an example of view-dependent refinement of the Buddha model, we obtain a throughput of roughly 800,000 rendered triangles per second. This video also illustrates the directionality of our anisotropic error metric, by animating the cube-like object from Figure 1. When the large, nearly flat faces of the cube are orthogonal to the view direction, they are coarsened significantly, while silhouettes such as the edges of the cube and faces close to tangent to the view direction are preserved. Figure 2 shows additional qualitative results.

**Disk and Memory Usage** Table 6 lists numerical results for our off-line method, including the number of triangles, grid size ($2^n$), timing results, effective triangle processing rate, and disk usage. Not included in this table is the maximum memory usage. Our method uses only a constant $O(1)$ amount of memory: 5 MB for the Linux machine, and 8 MB for the SGI (the difference is due to different size executables). The temporary disk usage of our method can be shown to be a constant multiple (roughly a factor of 5) of the size of the input mesh. While the theoretical usage is linear in both the size of the input and the output, the simplification phase often reduces the mesh to the extent that the overall temporary disk space is entirely dominated by the plane equation file (which is removed before the hierarchy construction begins) and any disk space used while sorting this file. As can be seen from the last column, the size of the octree output file is on average only 25% larger (in bytes per triangle) than the size of the input. For the sake of fairness, our triangle soup input format is not as efficient as the more common indexed mesh representation, which requires only half as much storage. The indexed mesh, on the other hand, is impractical for external memory algorithms since it requires random access.

**Execution Time** As can be seen from the table, for small octrees the total time is dominated by the external sort phase of the simplification. As the octree (and thus size of the output) grows larger, the output phase of the octree construction begins to dominate. Still, our memory insensitive algorithm is remarkably fast, and yields an effective triangle processing rate (measured as the size of the input over the total time) of roughly 20,000–50,000 triangles per second (tps). As a point of reference, El-Sana and Chiang [4] report a reduction rate of roughly 5,300 tps for their largest model, consisting of 1.2 million triangles. This model is considerably smaller than some of those simplified here, and it is not clear to what extent their method scales—an extrapolation of their results suggests that the speed of their method would steadily decline for increasing model size. The method proposed by Prince [21], while producing high quality meshes, yields about 1,000 tps for

| *model* | $T_{in}$ | $T_{out}$ | $n$ | *simp. time (%)* | | | *hier. time (%)* | | *total time* | $T_{in}/s$ | *disk usage (MB)* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | *read* | *sort* | *write* | *pull* | *push* | *(h:m:s)* | | *input* | *temp.* | *output* |
| Buddha | 1,087,716 | 62,346 | 7 | 19.3 | 44.6 | 20.3 | 0.3 | 15.5 | 21 | 51,345 | 37 | 180 | 2.6 |
| | | 204,766 | 8 | 17.5 | 30.1 | 13.6 | 0.7 | 37.3 | 32 | 33,685 | | 181 | 9.0 |
| | | 522,700 | 9 | 7.6 | 17.3 | 8.1 | 1.3 | 65.8 | 57 | 19,168 | | 186 | 26.1 |
| Blade | 28,246,208 | 129,214 | 7 | 23.3 | 48.8 | 21.4 | 0.0 | 6.5 | 13:02 | 36,109 | 970 | 4,823 | 5.1 |
| | | 507,104 | 8 | 21.3 | 34.4 | 19.9 | 0.1 | 24.3 | 14:15 | 33,025 | | 4,813 | 21.1 |
| St. Matthew | 372,963,401 | 3,012,996 | 10 | 24.0 | 43.0 | 23.2 | 0.0 | 10.5 | 3:56:30 | 26,284 | 12,805 | 64,011 | 125.6 |

Table 6: Numerical results for memory insensitive simplification and level-of-detail hierarchy construction. The timings are reported for the subphases *read* (lines 1–7), *sort* (line 8), and *write* (lines 9–12) of the simplification phase (see Table 1), and *pull* (lines 1–4) and *push* (lines 5–19) of the hierarchy construction phase (see Table 2). The Buddha data set was simplified on a Linux PC, while the other models were simplified on an SGI Onyx2.

the largest model used (11.4 million triangles). Meanwhile, his method requires more than 512 MB of RAM to simplify this model, whereas we use a constant 8 MB. The theoretical execution time of our off-line processing is $O(T_{in} + T_{out})$, assuming the external sort is implemented as a radix sort,[2] which suggests that our method scales well.

## 10 SUMMARY AND FUTURE WORK

We have described a method for constructing a level-of-detail hierarchy for large polygonal meshes. This method performs all computations on disk, and uses only a small, constant amount of RAM. The method requires temporary disk space linear in the size of the input and output, and runs in linear time. Even though virtually no RAM is used, our method executes one to two orders of magnitude faster than previous methods, and achieves a peak simplification rate of over 50,000 triangles per second. We have also presented compact data structures and a suitable error metric for performing out-of-core view-dependent refinement of the resulting mesh hierarchy. Our results show that we obtain interactive frame rates and a throughput of around 800,000 triangles per second using immediate mode rendering.

The work described in this paper has focused on the off-line simplification and hierarchy construction phases, while our run-time view-dependent component is currently rather primitive (although fairly general). We see considerable room for algorithmic improvement to the run-time system, such as the use of multithreading to decouple refinement and rendering, time-critical, priority-driven mesh updates (cf. [3]) to capitalize on frame-to-frame coherence, geomorphing to smooth out temporal popping artifacts, prefetching to improve the paging system, and fast culling to remove geometry that is either occluded or not within the view frustum. We believe that the regularity and hierarchical nature of the octree structure is particularly well suited for supporting fast culling. We also intend to investigate how to extend our error metric to guarantee the nesting condition in screen space, and how to account for the visual impact of simplification on the shading of the surface.

Finally, we envision that our off-line algorithms can be further enhanced. As noted in [18], the greatest potential for reducing the disk usage lies in using a compressed, perhaps quantized representation for the rather large plane equation file, which by far dominates the amount of temporary disk space used. We also see untapped potential in increasing the speed of our out-of-core method through parallelization. The locality of our data accesses and the octree partitioning of space and work naturally lend themselves to parallel execution.

## Acknowledgements

We would like to thank Stanford University and the Digital Michelangelo Project for providing the Buddha and St. Matthew data sets, and Kitware for the turbine blade model.

[2] The external sort `rsort` used in our implementation is a combination of radix and merge sort.

## References

[1] F. Bernardini, J. Mittleman, and H. Rushmeier. Case Study: Scanning Michelangelo's Florentine Pietà. SIGGRAPH 99 Course #8, Aug. 1999. URL http://www.research.ibm.com/pieta.

[2] J. Cohen, D. G. Aliaga, and W. Zhang. Hybrid Simplification: Combining Multi-Resolution Polygon and Point Rendering. *IEEE Visualization 2001*, 37–44. Oct. 2001.

[3] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. *IEEE Visualization '97*, 81–88. Nov. 1997.

[4] J. El-Sana and Y.-J. Chiang. External Memory View-Dependent Simplification. *Computer Graphics Forum*, 19(3):139–150, Aug. 2000.

[5] J. El-Sana and A. Varshney. Generalized View-Dependent Simplification. *Computer Graphics Forum*, 18(3):83–94, Sep. 1999.

[6] C. Erikson, D. Manocha, and W. V. Baxter III. HLODs for Faster Display of Large Static and Dynamic Environments. *2001 ACM Symposium on Interactive 3D Graphics*, 111–120. Mar. 2001.

[7] T. A. Funkhouser and C. H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Proceedings of SIGGRAPH 93*, 247–254. Aug. 1993.

[8] M. Garland. *Quadric-Based Polygonal Surface Simplification.* Ph.D. thesis, Carnegie Mellon University, May 1999.

[9] M. Garland and P. S. Heckbert. Surface Simplification Using Quadric Error Metrics. *Proceedings of SIGGRAPH 97*, 209–216. Aug. 1997.

[10] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 3rd edn., 1996.

[11] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97*, 189–198. Aug. 1997.

[12] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *IEEE Visualization '98*, 35–42. Oct. 1998.

[13] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. *Proceedings of SIGGRAPH 2000*, 131–144. Jul. 2000. URL http://graphics.stanford.edu/projects/mich.

[14] J. Linderman. `rsort` man page, Apr. 1996. Revised Jun. 2000.

[15] P. Lindstrom. Out-of-Core Simplification of Large Polygonal Models. *Proceedings of SIGGRAPH 2000*, 259–262. Jul. 2000.

[16] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. *Proceedings of SIGGRAPH 96*, 109–118. Aug. 1996.

[17] P. Lindstrom and V. Pascucci. Visualization of Large Terrains Made Easy. *IEEE Visualization 2001*, 363–370. Oct. 2001.

[18] P. Lindstrom and C. T. Silva. A Memory Insensitive Technique for Large Model Simplification. *IEEE Visualization 2001*, 121–126. Oct. 2001.

[19] D. Luebke and C. Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH 97*, 199–208. Aug. 1997.

[20] D. Luebke and B. Hallen. Perceptually-Driven Simplification for Interactive Rendering. *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 223–234. Jun. 2001.

[21] C. Prince. *Progressive Meshes for Large Models of Arbitrary Topology.* Master's thesis, University of Washington, 2000.

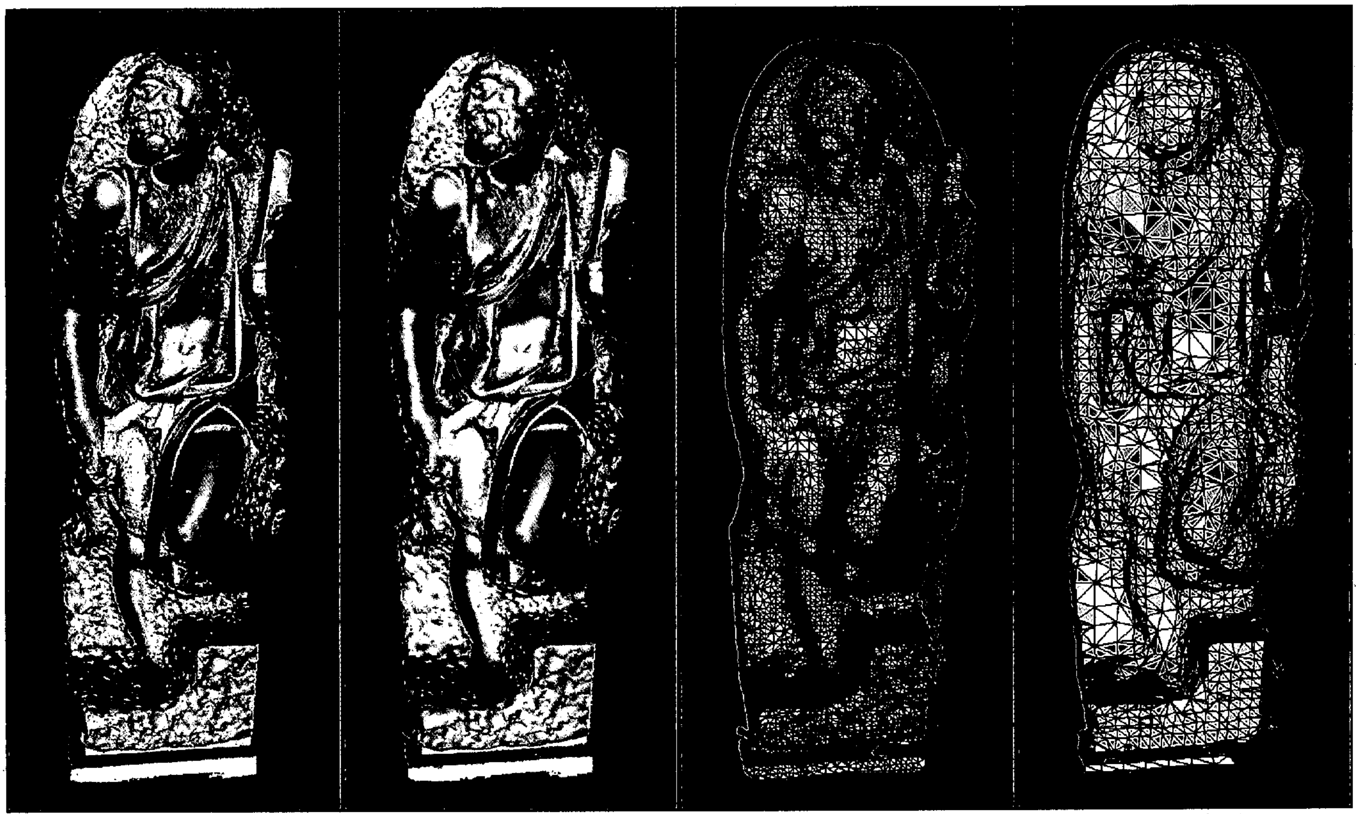
[22] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. *Modeling in Computer Graphics*, 455–465. Springer-Verlag, 1993.

[23] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. *Proceedings of SIGGRAPH 2000*, 343–352. Jul. 2000.

[24] E. Shaffer and M. Garland. Efficient Adaptive Simplification of Massive Meshes. *IEEE Visualization 2001*, 127–134. Oct. 2001.

[25] K. Shoemake. Quaternions and 4x4 Matrices. *Graphics Gems II*, 351–354. Academic Press, 1991.

[26] J. C. Xia and A. Varshney. Dynamic View-Dependent Simplification for Polygonal Models. *IEEE Visualization '96*, 327–334. Oct. 1996.



(a) 2,860,000 triangles. (b) 1,216,000 triangles. (c) 166,000 triangles. (d) 15,000 triangles.

Figure 2: View-dependent renderings of the St. Matthew data set for various error thresholds. The original model contains 373 million triangles.